Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2005-051
MIT-LCS-TR-998

August 8, 2005

Implementing Probabilistically Checkable Proofs of Proximity

Arnab Bhattacharyya

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

CSAIL

Implementing Probabilistically Checkable Proofs of Proximity

Arnab Bhattacharyya*
MIT Computer Science and Artificial Intelligence Lab
abhatt@csail.mit.edu

Abstract

In this paper, we describe a proof-of-concept implementation of the probabilistically checkable proof of proximity (PCPP) system described by Ben-Sasson and Sudan in [BSS05]. In particular, we implement a PCPP prover and verifier for Reed-Solomon codes; the prover converts an evaluation of a polynomial on a linear set into a valid PCPP, while the verifier queries the evaluation and the PCPP to check that the evaluation is close to a Reed-Solomon codeword. We prove tight bounds on the various parameters associated with the prover and verifier and describe some interesting programmatic issues that arise during their implementation.

1 Introduction

A probabilistically checkable proof (PCP) system specifies a format for writing proofs that can be verified efficiently by querying only a few bits. Formally, a PCP system consists of an input string, a source of random bits, a proof string, and a probabilistic polynomial-time Turing machine called the verifier. The verifier has random access to the proof; given an address of a location in the proof, the verifier can query that location in the proof as a single oracle operation. A PCP verifier V with perfect completeness and soundness $s(n)$ for the language L satisfies the following conditions:

- For every input x in L, there is a proof Π such that V accepts with probability 1.
- For every input x not in L and for every proof Π, V accepts with probability less than $s(n)$.

Furthermore, a language L is said to be in PCP[$r(n)$,$q(n)$] if there is a PCP verifier for L that on each input of size n uses at most $r(n)$ random bits and queries at most $q(n)$ bits of the proof. The celebrated PCP Theorem states that for any language in NP, there exists a PCP verifier with soundness 1/2 that uses $O(\log n)$ random bits and queries $O(1)$ bits of the proof. Hence, the size of the proof needed by the verifier is $2^{O(\log n)} = poly(n)$, polynomially larger than the size of the NP-witness.

Subsequently, much work has been done in trying to reduce the length of the proof and to make its constructions simpler. The length of the proof is relevant to applications of PCP theory in cryptography and to constructions of locally testable codes (LTCs). Moreover, there is the possibility that a PCP system with short proof size could form the basis for a semantic analog of error-correcting codes. Simplifying the proof construction is also important for this reason. Some progress toward these goals were made in [BSS05] where Ben-Sasson and Sudan showed that there exist probabilistically checkable proofs for verifying

[*] This work was supported by a REU supplement to NSF ITR Award CCR-0312575. Any opinions, findings and conclusions or recommendations expressed in this report are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

satisfiability of circuits of size n of length $n \cdot poly(\log n)$, with the verifier querying $poly(\log n)$ bits of the proof. Moreover, the construction of the proof is significantly simpler than in previous PCP constructions. Their Theorem 1 states:

> **Theorem 1 ([BSS05], Theorem 1)**: SAT has a PCP verifier that on inputs of length n tosses $\log(n \cdot poly(\log n))$ coins, makes $poly(\log n)$ queries to a proof oracle of length $n \cdot poly(\log n)$, runs in time $n \cdot poly(\log n)$ and has perfect completeness and soundness at most $\frac{1}{2}$.

This PCP construction involves the construction of *probabilistically checkable proofs of proximity* (PCPPs) for Reed-Solomon codes. PCPPs provide an even stronger restriction on the verifier's computation, compared to the standard PCP model. Whereas a PCP verifier has unrestricted access to the input string but is restricted to making only a few queries to the proof, a PCPP has restricted access to both the input and the proof. Formally:

> **Definition 1 (PCPP)** A set $C \subseteq \Sigma^n$ has a *probabilistically checkable proof of proximity* over alphabet Σ of length $\ell(n)$ with query complexity $q(n)$, perfect completeness and soundness $s(\cdot, n)$ if there exists a verifier V with oracle access to a pair $(x, \pi) \in \Sigma^{n+\ell(n)}$ such that V tosses $r(n)$ coins, makes $q(n)$ queries into (x, π) and accepts or rejects as follows:
>
> – If $x \in C$, then $\exists \pi \in \Sigma^{\ell(n)}$ such that verifier accepts (x, π) with probability 1.
>
> – If $\Delta(x, C) \geq \delta$, then $\forall \pi \in \Sigma^{\ell(n)}$, verifier rejects (x, π) with probability at least $s(\delta, n)$.

[BSS05] provides efficient PCPPs for Reed-Solomon codes, which are defined next:

> **Definition 2 (RS-Codes)** The Reed-Solomon code of degree d over a field $\mathcal{F}$ evaluated at $S \subseteq \mathcal{F}$ is defined as $\mathrm{RS}(\mathcal{F}, S, d)$=$\{\langle P(z)\rangle_{z \leftarrow S} : P(z) = \sum_{i=0}^{d-1} a_i z^i, a_i \in \mathcal{F}\}$, where $\langle P(z)\rangle_{z \leftarrow S}$, the evaluation table of P over S, is the sequence $\langle P(s) : s \in S\rangle$ and S has some canonical ordering to make it a sequence.

The primary result in [BSS05] regarding PCPPs for RS-codes that we are concerned with is the following:

> **Theorem 2 ([BSS05], Theorem 4)** There exists a universal constant $c \geq 1$ such that for every field $\mathcal{F}$ of characteristic two, every linear $S \subseteq \mathcal{F}$ with $|S| = n$ and every $d \leq n$, the Reed-Solomon code $\mathrm{RS}(\mathcal{F}, S, d)$ has a PCPP over alphabet $\mathcal{F}$ with proof length $l(n) \leq n \log^c n$, randomness $r(n) \leq \log n + c \log\log n$, query complexity $q(n) = O(1)$, and soundness $s(\delta, n) \geq \delta / \log^c n$.

In this paper, we describe an actual implementation of this PCPP system for Reed-Solomon codes. Specifically, the following two programs are implemented:

1. A prover that receives as input a description of the field $\mathcal{F} = GF(2^l)$, a basis $(b_1, \ldots, b_k)$ for $L \subseteq \mathcal{F}$, a degree parameter d and a polynomial $P : L \to \mathcal{F}$ of degree less than d, and that outputs a PCPP which is supposed to prove that $\langle P(z)\rangle_{z \leftarrow L}$ is in RS($\mathcal{F}$,L,d).

2. A verifier that receives as input a description of the field $\mathcal{F} = GF(2^l)$, a basis $(b_1, \ldots, b_k)$ for $L \subseteq \mathcal{F}$, a degree parameter d and oracle access to a purported RS-codeword $p : L \to \mathcal{F}$ and its purported PCPP π, and that accepts or rejects based on the proximity of p to RS($\mathcal{F}$,L,d).

In the following, we detail these implementations and provide some tight bounds on the various complexity parameters associated with the PCPP system. These results establish that the constants associated with the PCPP size are not at all large and, so, could perhaps motivate the use of probabilistically checkable proofs in real-life as analogs to error-correcting codes.

2 Implementation of the PCPP system

The most basic operations in constructing and verifying the probabilistically checkable proofs of proximity described in [BSS05] are addition and multiplication in fields of characteristic two, extension fields of $GF(2)$. To do these operations efficiently while maintaining a proper programmatic abstraction, I used the excellent C++ library NTL, developed by Victor Shoup [Sho]. NTL is a high-quality and portable C++ library providing an efficient programmatic interface for computations over finite fields. Our PCPP prover and verifier programs are implemented as dynamically-linked C++ libraries with dependencies on the base NTL library. Thus, users of our implementation can link to our prover and verifier modules to create a valid PCPP and verify provided PCPPs respectively.

NTL represents elements of the field $GF(2^l)$ as polynomials in $GF(2)[x]$ modulo an irreducible polynomial P of degree l. Hence, in the following, I will view field elements as vectors from the (additive) vector space $GF(2)^l$. For the prover to provide a proof acceptable to the verifier, it must use the same irreducible polynomial P as the verifier. Also, both must sequence the field elements in the same order, and both must use the same bases elements for any subspaces of $\mathcal{F}$ that are considered.

2.1 Evaluation and Interpolation of Polynomials

The following two problem need to be solved repeatedly while constructing and verifying our PCPPs:

- (**Evaluation**) Given a finite field $\mathcal{F}$ of characteristic 2, coefficients $c_0, \ldots, c_{n-1} \in \mathcal{F}$ and linearly independent elements $e_1, \ldots, e_k \in \mathcal{F}$ with $n = 2^k$, compute the set $\{(\alpha, p(\alpha)) | \alpha \in \text{span}(e_1, \ldots, e_k)\}$ where $p(x) = \sum_{i=0}^{n-1} c_i x^i$.
- (**Interpolation**) Given a finite field $\mathcal{F}$ of characteristic 2, linearly independent elements $e_1, \ldots, e_k \in \mathcal{F}$ and the set $\{(\alpha, p_\alpha) | \alpha \in \text{span}(e_1, \ldots, e_k)\}$, compute coefficients $c_0, \ldots, c_{n-1} \in \mathcal{F}$ such that $p_\alpha = \sum_{i=0}^{n-1} c_i \alpha^i$ for all $\alpha \in \text{span}(e_1, \ldots, e_k)$.

Both can be achieved with $O(n \log^2 n)$ field operations[1] using a Fast Fourier Transform method. Here, I will describe the solution to the interpolation problem; the solution to the evaluation problem is very similar although not identical. The key ideas behind the interpolation algorithm are in the lemmas below:

> **Lemma 1** Given $e_1, \ldots, e_k \in \mathcal{F}$, there exists a monic quadratic $q(x)$ such that for every $\alpha \in \text{span}(e_1, \ldots, e_{k-1})$, $q(\alpha) = q(\alpha + e_k)$. Also, there exists vectors $e'_1, \ldots, e'_{k-1} \in \mathcal{F}$ such that for all $\alpha \in \text{span}(e_1, \ldots, e_k)$, $q(\alpha) \in \text{span}(e'_1, \ldots, e'_{k-1})$. Further, q and $e'_1, \ldots, e'_{k-1}$ can be computed in time $O(k)$.
>
> **Proof** Let $q(x) = x^2 - e_k \cdot x$ and let $e'_i = q(e_i)$ for $1 \leq i \leq k-1$. Note that $q(x+y) = q(x) + q(y)$ since we are in a field of characteristic 2. So, because $q(e_k) = 0$, the first assertion is true. The second assertion holds since if $\alpha = \sum_{i=1}^{k} \lambda_i e_i$ with $\lambda_i \in GF(2)$, then $q(\alpha) = \sum_{i=1}^{k} q(\lambda_i e_i) = \sum_{i=1}^{k-1} \lambda_i q(e_i)$.□
>
> **Lemma 2** Given the set $\{(\alpha, p_\alpha) | \alpha \in \text{span}(e_1, \ldots, e_k)\}$ and the monic degree 2 polynomial q and the elements $\{e'_i\}_{i=1}^{k-1}$ from Lemma 1, there exist sets $\{(\alpha', p^0_{\alpha'}) | \alpha' \in \text{span}(e'_1, \ldots, e'_{k-1})\}$ and $\{(\alpha', p^1_{\alpha'}) | \alpha' \in \text{span}(e'_1, \ldots, e'_{k-1})\}$ such that $p_\alpha = p^0_{q(\alpha)} + \alpha \cdot p^1_{q(\alpha)}$ for all $\alpha \in \text{span}(e_1, \ldots, e_k)$. Moreover, the two sets can be computed in time $O(n)$.
>
> **Proof** Note that from the properties of q in Lemma 1, we want the two sets to be such that for all $\alpha \in \text{span}(e_1, \ldots, e_k)$, $p_\alpha = p^0_{q(\alpha)} + \alpha \cdot p^1_{q(\alpha)}$ and $p_{\alpha+e_k} = p^0_{q(\alpha)} + (\alpha + e_k) \cdot p^1_{q(\alpha)}$. So, $p^1_{q(\alpha)} = e_k^{-1} \cdot (p_{\alpha+e_k} - p_\alpha)$. Also, then, $p^0_{q(\alpha)} = p_\alpha - \alpha \cdot p^1_{q(\alpha)} = p_\alpha - e_k^{-1} \cdot (p_{\alpha+e_k} - p_\alpha)$. Assuming constant-time access to p_α, these calculations can be done for all $\alpha' = q(\alpha) \in \text{span}(e'_1, \ldots, e'_{k-1})$ in time $O(n)$.□

[1] Field operations take $O(\log |\mathcal{F}|)$ bit operations and will be taken to have unit time cost.

Lemma 3 Given coefficients of two polynomials $p^0(x)$ and $p^1(x)$ of degree less than $n/2$ and any monic degree 2 polynomial $q(x)$, then there exists a polynomial $p(x)$ of degree less than n such that $p(x) = p^0(q(x)) + x \cdot p^1(q(x))$. Moreover, the coefficients of p can be computed in time $O(n \log n)$.

Proof The existence statement is clear. We just have to give an efficient algorithm to find the coefficients of $p(x)$. First of all, write $p^0(z) = b^0(z) + z^{n/4}a^0(z)$ and $p^1(z) = b^1(z) + z^{n/4}a^1(z)$, where a^0, a^1, b^0 and b^1 are polynomials of degree less than $n/4$. Recursively, we can find the coefficients of the polynomials $a(x)$ and $b(x)$, where $a(x) = a^0(q(x)) + x \cdot a^1(q(x))$ and $b(x) = b^0(q(x)) + x \cdot b^1(q(x))$; $a(x)$ and $b(x)$ have degrees less than $n/2$. Now, $p(x) = b(x) + q(x)^{n/4} \cdot a(x)$. Since n is a power of 2, if $q(x) = x^2 + cx + d$, then $q(x)^{n/4} = x^{n/2} + c^{n/4}x^{n/4} + d^{n/4} = x^{n/2} + c'x^{n/4} + d'$. Writing $a(x) = \sum_{i=0}^{n/2-1} \alpha_i x^i$ and $b(x) = \sum_{i=0}^{n/2-1} \beta_i x^i$, we can see that $p(x) = \sum_{i=0}^{n/4-1}(d'\alpha_i + \beta_i)x^i + \sum_{i=n/4}^{n/2-1}(d'\alpha_i + c'\alpha_{i-n/4} + \beta_i)x^i + \sum_{i=n/2}^{3n/4-1}(\alpha_{i-n/2} + c'\alpha_{i-n/4})x^i + \sum_{i=3n/4}^{n-1} \alpha_{i-n/2}x^i$. Thus, we can get the coefficients of $p(x)$ from the coefficients of $a(x)$ and $b(x)$ in $O(n)$ time, and so the total time for the recursion is $O(n \log n)$ as claimed. □

Given these lemmas, the interpolation algorithm follows:

```
**InvFFT-Additive**$(e_1, \ldots, e_k, \{(\alpha, p_\alpha) | \alpha \in \text{span}(e_1, \ldots, e_k)\})$
  1. Compute $q(x), e'_1, \ldots, e'_{k-1}$ as by Lemma 1.
  2. Compute $\{(\alpha', p^0_{\alpha'}) | \alpha' \in \text{span}(e'_1, \ldots, e'_{k-1})\}$ and $\{(\alpha', p^1_{\alpha'}) | \alpha' \in \text{span}(e'_1, \ldots, e'_{k-1})\}$ as by Lemma 2.
  3. Compute $p^0(x) = \textbf{InvFFT-Additive}(e'_1, \ldots, e'_{k-1}, \{(\alpha', p^0_{\alpha'}) | \alpha' \in \text{span}(e'_1, \ldots, e'_{k-1})\})$.
  4. Compute $p^1(x) = \textbf{InvFFT-Additive}(e'_1, \ldots, e'_{k-1}, \{(\alpha', p^1_{\alpha'}) | \alpha' \in \text{span}(e'_1, \ldots, e'_{k-1})\})$.
  5. Compute $p(x)$ from $p^0(x)$ and $p^1(x)$ as by Lemma 3.
```

The running time for the algorithm is $O(n \log^2 n)$ because each recursion halves the span of the bases elements. During implementation, a choice must be made as to the data structure to be used in storing the evaluation table of a polynomial. Although in the proof of Lemma 2, we assumed that we need constant-time to retrieve p_α given α, our implementation uses an associative data container, based on a red-black tree which has a $O(\log n)$ access time. It can be checked that this does not affect[2] the asymptotic running time for the interpolation and evaluation algorithms. (The choice to use a logarithmic-time container instead of a constant-time container was made merely for convenience reasons; the C++ Standard Template Library provides the `map` data type, while there is no corresponding type for a hash table.)

The C++ data structure declarations and function signatures associated with evaluation and interpolation of polynomials are shown in Listing 1. The code listing shows the two most important NTL types that are used in the PCPP implementation. `GF2E` is the type of an element in an extension field of $GF(2)$, and `GF2EX` is the type of a polynomial with coefficients of type `GF2E`. Before its first use, `GF2E` needs to be initialized with an irreducible polynomial in $GF(2)[x]$ to specify the extension of $GF(2)$. More details regarding the NTL programmatic interface to finite field computations can be found at `http://www.shoup.net`.

Listing 1: Evaluating and interpolating polynomials on fields of characteristic two

```
/** Evaluation table of a function f on elements of a field of characteristic 2.
  */
struct eval_table{
  // Stores pairs <x,f(x)>
```

[2] Comparison of two field elements in traversing the red-black tree takes $O(\log |\mathcal{F}|)$ time, same as that for any other field operation. As before, we take field operations to have unit time cost.

```
  // (ltGF2E is the comparison operator on field elements)
  map<GF2E,GF2E,ltGF2E> evalmap;

  // Given x, return f(x), assuming <x,f(x)> is in evalmap.
  // Running time: O(log n)
  GF2E query(const GF2E& x) const;

  // Store the pair <x,y>
  // Running time: O(log n)
  void insert(const GF2E& x, const GF2E& y);

  // Clear the evaluation table
  // Running time: O(1)
  void clear();
};

/** Store in <table> the evaluation of the polynomial <poly> on the set of
  * n=2^k field elements spanned by the k elements in <bases>.
  * Running time: O(n (log n)^2)
  */
void eval_poly(eval_table& table, const GF2EX& poly, const vec_GF2E& bases);

/** Make <poly> the polynomial interpolated from <table>, the
  * evaluation table of a function at each element spanned by <bases>.
  * Running time: O(n (log n)^2)
  */
void interpolate_poly(GF2EX& poly, const eval_table& table, const vec_GF2E& bases);
```

2.2 The Prover

In this section, I will detail the implementation of the PCPP prover, the program that, given a polynomial over a field $\mathcal{F}$ of degree less than d and a subspace $L \subseteq \mathcal{F}$, constructs a valid probabilistically checkable proof of proximity that shows that the polynomial's evaluation table over L is in RS($\mathcal{F}$,L,d). The algorithms that appear in this section and the next are taken from [BSS04], the full version of the conference paper by Ben-Sasson and Sudan.

Throughout this paper, we will only consider the case when d is fixed to be $|S|/8$. As is shown in [BSS04], the more general case can be reduced to a sequence of these special PCPPs. It is convenient to think of the proof, not as a string of bits, but as an oracle that can be queried; the advantage of this viewpoint will become very apparent when we describe the verifier. The basic idea of the PCPP construction is that we convert a univariate polynomial of degree less than $n/8$ into a bivariate polynomial of degree less than $\sqrt{n}$ in each variable and then we invoke the Polischuk-Spielman analysis from [PS94] to reduce testing of bivariate polynomials to testing of univariate polynomials of approximately the same degree. To describe the proof more precisely, we will introduce the same notation as that used in [BSS04]. Throughout, assume that we are given a specific set of bases $(b_1, \ldots, b_k)$ for a linear subspace L of the field and that $n = 2^k = |L|$. Define the following:

- $\tilde{L}_0 = \mathrm{span}(b_1, \ldots, b_{\lfloor k/2 \rfloor})$
- $L_0 = \mathrm{span}(b_1, \ldots, b_{\lfloor k/2 \rfloor + 2})$
- $\tilde{L}_1 = \mathrm{span}(b_{\lfloor k/2 \rfloor + 1}, \ldots, b_k)$

- $q(x) = \prod_{\alpha \in \tilde{L}_0}(x - \alpha)$
- $L_1 = \mathrm{span}(q(b_{\lfloor k/2 \rfloor + 1}), \ldots, q(b_k))$
- $A_{\tilde{\beta}} = \{\tilde{\beta} + \alpha | \alpha \in \tilde{L}_0\}$, the affine shift of $\tilde{L}_0$ by $\tilde{\beta}$
- For $\tilde{\beta} \in \tilde{L}_1$, $L_{\tilde{\beta}} = \begin{cases} \mathrm{span}(L_0, b_{\lfloor k/2 \rfloor + 3}) & \text{if } \tilde{\beta} \in \mathrm{span}(b_{\lfloor k/2 \rfloor + 1}, b_{\lfloor k/2 \rfloor + 2}) \\ \mathrm{span}(L_0, \tilde{\beta}) & \text{otherwise} \end{cases}$
- $T = \{(\gamma, q(\gamma)) | \gamma \in L\}$

Next, we make a few observations that the reader can easily verify to follow directly from the above definitions. Firstly, $q(x)$ is a $GF(2)$-linear map with $\tilde{L}_0$ as its kernel (see Proposition 8 in [BSS04]). Secondly, for all $\tilde{\beta} \in \tilde{L}_1$, $|L_{\tilde{\beta}}| = 4|L_0| = 8|\tilde{L}_0|$ from the definition of $L_{\tilde{\beta}}$. Thirdly, it is clear that for all $\tilde{\beta} \in \tilde{L}_1$, $L_{\tilde{\beta}}$ is a linear set while $A_{\tilde{\beta}} \subseteq L_{\tilde{\beta}}$ is not linear unless $\tilde{\beta} = 0$. Finally, note that

$$T = \bigcup_{\tilde{\beta} \in \tilde{L}_1} A_{\tilde{\beta}} \times q(\tilde{\beta})$$

which follows from the fact that q is a linear transformation with kernel $\tilde{L}_0$.

Using the above notation, the structure of the Reed-Solomon PCP of proximity oracle is:

Definition 3 (**[BSS04], Definition 4**) The proof oracle for a codeword of the RS-code RS($GF(2^l)$,L,$|L|/8$) is defined by induction on $k = \dim(L)$. If $k \leq 6$, then it is empty. Otherwise, the proof is a pair $\pi = \{f, \Pi\}$ where f is a partial bivariate function over partial domain $S \subset GF(2^l) \times GF(2^l)$ and Π is a sequence of PCPPs for RS-codes over smaller linear spaces.

Partial domain S: Let $S_{\tilde{\beta}} = L_{\tilde{\beta}} \times \{q(\tilde{\beta})\}$ and let $T = \{(\gamma, q(\gamma)) | \gamma \in L\}$. Then

$$S = \left(\bigcup_{\tilde{\beta} \in \tilde{L}_1} S_{\tilde{\beta}} \right) - T = \bigcup_{\tilde{\beta} \in \tilde{L}_1} ((L_{\tilde{\beta}} - A_{\tilde{\beta}}) \times \{q(\tilde{\beta})\})$$

Auxiliary proofs Π: For each $\tilde{\beta} \in \tilde{L}_1$ and $\beta = q(\tilde{\beta}) \in L_1$, Π has one PCPP for an RS codeword over $L_{\tilde{\beta}}$ of degree $|L_{\tilde{\beta}}|/8$, denoted $\pi^{\leftrightarrow}_{\tilde{\beta}}$. For each $\alpha \in L_0$, Π includes a PCPP for an RS codeword over L_1 of degree $|L_0|/8$, denoted $\pi^{\updownarrow}_{\alpha}$. Formally,

$$\Pi = \{\pi^{\leftrightarrow}_{\tilde{\beta}} | \beta \in L_1\} \cup \{\pi^{\updownarrow}_{\alpha} | \alpha \in L_0\}$$

The C++ declaration of the PCPP object, shown in Listing 2, reflects the recursive structure of the proof described above.

Listing 2: Declaration of the PCPP data type

```
/** Analog of eval_table for a bivariate polynomial
  */
struct biv_eval_table {
  map<GF2E,eval_table,ltGF2E> evalmap;

  GF2E query(const GF2E& x, const GF2E& y) const;
  void insert(const eval_table& xvals, const GF2E& y);
  void clear();
};

/** Representation of a PCPP oracle for RS-codes
  */
struct poly_oracle{
  // Evaluation of f on S
  biv_eval_table eval;

  // Pointers to the auxiliary proofs Π
  vector<poly_oracle*> proof;

  // Pointer to additional PCPPs
  poly_oracle* next;
};
```

Now, having specified the form of a correct PCPP in Definition 3, we need to specify its contents, the bivariate polynomial f and the auxiliary proofs Π.

- **Construction of f:** Given the polynomials p and q, construct the unique bivariate polynomial $Q(x, y)$ with $\deg_x(Q) < \deg(q)$ and $\deg_y(Q) < \lfloor \deg(p)/\deg(q) \rfloor$ such that $p(x) = Q(x, q(x))$ for all $x \in L$. That such a Q exists and is unique is given by Proposition 7 in [BSS04], and the algorithm to compute it is discussed below in 2.2.1. In our case, p is of degree $n/8$ while q is roughly of degree $\sqrt{n}$; so, Q is roughly of degree $\sqrt{n}$ in x and $\sqrt{n}/8$ in y. Now, define $f(\alpha, \beta) = Q(\alpha, \beta)$ for all $(\alpha, \beta) \in S$. This is the bivariate function whose evaluation table over S is provided in the PCPP.
- **Construction of Π:** Denote by $\hat{p} : T \to \mathcal{F}$ the bivariate polynomial defined by $\hat{p}(x, q(x)) = p(x)$ for all $x \in L$ [3]. Then let $\hat{f}$ be the function that agrees with f on S and $\hat{p}$ on T. Also define $\hat{f}|^{\leftrightarrow}_{\beta} : \{\alpha | (\alpha, \beta) \in S \cup T\} \to \mathcal{F}$ as $\hat{f}|^{\leftrightarrow}_{\beta}(\alpha) = \hat{f}(\alpha, \beta)$. Similarly, define $\hat{f}|^{\updownarrow}_{\alpha} : \{\beta | (\alpha, \beta) \in S \cup T\} \to \mathcal{F}$ as $\hat{f}|^{\updownarrow}_{\alpha}(\beta) = \hat{f}(\alpha, \beta)$. It is fairly easy to verify (see Proposition 10 in [BSS04]) that for $\tilde{\beta} \in \tilde{L}_1$ and $\beta = q(\tilde{\beta})$, $\{\alpha | (\alpha, \beta) \in S \cup T\} = L_{\tilde{\beta}}$ and that for $\alpha \in L_0$, $\{\beta | (\alpha, \beta) \in S \cup T\} = L_1$. Then for $\tilde{\beta} \in \tilde{L}_1$ and $\beta = q(\tilde{\beta})$, $\pi^{\leftrightarrow}_{\tilde{\beta}}$ is the PCPP proving that $\hat{f}|^{\leftrightarrow}_{\beta}$ is a codeword in $\mathrm{RS}(\mathcal{F}, L_{\tilde{\beta}}, |L_{\tilde{\beta}}|/8)$, and for $\alpha \in L_0$, $\pi^{\updownarrow}_{\alpha}$ is the PCPP proving that $\hat{f}|^{\updownarrow}_{\alpha}$ is a codeword in $\mathrm{RS}(\mathcal{F}, L_1, |L_1|/8)$.

This same description in C++ code is given in Listing 3.

Listing 3: Construction of Reed-Solomon PCPPs

```
// d = |L|/8
void ReedSolomon_PCPP(poly_oracle& pcpp, const GF2EX& poly, const vec_GF2E& L_bases){
  vec_GF2E L00_bases, L10_bases, L0_bases, L1_bases, Lbeta_bases;
  long k = L_bases.length(), i, j;
  GF2EX q, frow, fcol;
  vec_GF2EX f;
  vec_GF2E L0_span, L10_span, Lbeta_span, L1_span;
  GF2E beta0, beta, tmp;
  eval_table coleval, roweval;
  biv_eval_table bioracle;

  if(L_bases.length() > 6){  // 6 because floor(k/2)+3<k for k>=7
```

[3] Notice that a verifier does not need a separate evaluation table for $\hat{p}$ because it can simply use the provided evaluation table for p; separately evaluating $\hat{p}$ and f is crucial to proving the soundness of the verifier.

```
// get the bases for $\tilde{L}_0$, $L_0$ and $\tilde{L}_1$.
get_L00_bases(L00_bases, L_bases);
get_L0_bases(L0_bases, L_bases);
get_L10_bases(L10_bases, L_bases);

// get q of degree approximately $\sqrt{n}$
LinearizedPoly(q, L00_bases);

// get the bases for $L_1$
get_L1_bases(L1_bases, L10_bases, q);

// get all elements in $\tilde{L}_1$, $L_0$ and $L_1$ for later use
get_span(L10_span, L10_bases);
get_span(L0_span, L0_bases);
get_span(L1_span, L1_bases);

// get the bivariate polynomial $f$
create_bivariate(f, poly, q, L10_span); // given in Listing 4

// evaluate the bivariate polynomial $f$ on $S \cup T$
for(i=0; i<L10_span.length(); i++){
  beta0 = L10_span[i]; // for each $\tilde{\beta} \in \tilde{L}_1$

  // get the bases for $L_{\tilde{\beta}}$
  get_Lbeta_bases(Lbeta_bases, beta0, L_bases);

  // Find $f(\alpha, q(\tilde{\beta}))$ for all $\alpha \in L_{\tilde{\beta}}$, i.e. the $q(\tilde{\beta})$-row of $S \cup T$
  roweval.clear();
  eval_poly(roweval, f[i], Lbeta_bases);

  biorcale.insert(roweval, EvalLinearizedPoly(q, beta0));
}

// Construct the auxiliary proofs $\Pi$
vector<poly_oracle*> proofs(L10_span.length() + L0_span.length());

// Construct the proofs $\pi_{\tilde{\beta}}^{\leftrightarrow}$ for all $\tilde{\beta} \in \tilde{L}_1$ with $\beta = q(\tilde{\beta})$
for(i=0; i<L10_span.length(); i++){
  proofs.at(i) = new poly_oracle;
  get_Lbeta_bases(Lbeta_bases, L10_span[i], L_bases);

  // proof that $\hat{f}|_{\tilde{\beta}}^{\leftrightarrow}$ is in RS($\mathcal{F}, L_{\tilde{\beta}}, |L_{\tilde{\beta}}|/8$)
  ReedSolomon_PCPP(*proofs.at(i), f[i], Lbeta_bases);
}

// Construct the proofs $\pi_{\alpha}^{\updownarrow}$ for all $\alpha \in L_0$
for(i=0; i<L0_span.length(); i++){
  coleval.clear();
  proofs.at(i+L1_span.length()) = new poly_oracle;
  for(j=0; j<L1_span.length(); j++){
    coleval.insert(L1_span[j], bioracle.query(L0_span[i], L1_span[j]));
  }
```

```
      interpolate_poly(fcol, coleval, L1_bases);

      // proof that $\hat{f}|^{\updownarrow}_{\alpha}$ is in RS($\mathcal{F}, L_1, |L_1|/8$)
      ReedSolomon_PCPP(*proofs.at(i+L1_span.length()), fcol, L1_bases);
    }
    pcpp.eval = bioracle;
    pcpp.proof = proofs;
  }
  pcpp.next = 0;
  return;
}
```

2.2.1 Running Time of the Prover

Let $T(n)$ denote the running time of the algorithm shown in Listing 3 for $n = |L|$. Let $T_f(n)$ denote the time required to find the bivariate polynomial f. Then from inspection of the algorithm, it can be seen that asymptotically:

$$
\begin{aligned}
T(n) &= \begin{cases} T_f(n) + O(2^{\lceil k/2 \rceil}(8 \cdot 2^{\lfloor k/2 \rfloor} \log^2(8 \cdot 2^{\lfloor k/2 \rfloor}))) + 2^{\lceil k/2 \rceil} \cdot T(8 \cdot 2^{\lfloor k/2 \rfloor}) + 4 \cdot 2^{\lfloor k/2 \rfloor} \cdot T(2^{\lceil k/2 \rceil}) & \text{if } k > 6 \\ 0 & \text{if } k \leq 6 \end{cases} \\
&= \begin{cases} T_f(n) + O(n \log^2(n)) + 2^{\lceil k/2 \rceil} \cdot T(8 \cdot 2^{\lfloor k/2 \rfloor}) + 4 \cdot 2^{\lfloor k/2 \rfloor} \cdot T(2^{\lceil k/2 \rceil}) & \text{if } k > 6 \\ 0 & \text{if } k \leq 6 \end{cases}
\end{aligned}
$$

where $k = \log(n)$. So, we need to find $T_f(n)$ in order to solve the recurrence above for $T(n)$. Recall that f is the restriction to S of a bivariate polynomial Q which satisfies the relationship, $Q(x, q(x)) = p(x)$, on T and which has $\deg_x(Q) < \deg(q)$ and $\deg_y(Q) < \lfloor \deg(p)/\deg(q) \rfloor$. Also, notice from Listing 3 that we represent a bivariate polynomial over x and y as a sequence of univariate polynomials over x, one for each value of y in the domain. The algorithm that we use for calculating Q uses division over the ring of bivariate polynomials. Note that if we fix a lexicographic ordering on terms with $x > y$, then dividing $p(x)$ by $q(x) - y$, we obtain

$$p(x) = Q'(x, y) \cdot (q(x) - y) + Q(x, y)$$

It can be easily checked that this remainder $Q(x, y)$ has the requisite properties. For our representation, we want to evaluate $Q(x, \beta)$ for all $\beta \in L_1$. The following lemma asserts that $Q(x, \beta)$ is the remainder after the univariate division of $p(x)$ by $q(x) - \beta$.

> **Lemma 4:** Let $\mathcal{F}[x, y]$ be the ring of bivariate polynomials with the lexicographic ordering $x > y$ on terms. Suppose $f \in \mathcal{F}[x]$ and $g \in \mathcal{F}[x, y]$. Also, $g(x, y) \equiv m(x) + n(y)$ where $m \in \mathcal{F}[x]$ and $n \in \mathcal{F}[y]$. Let $h(x, y)$ be the remainder after dividing $f(x)$ by $g(x, y)$. Then, for any $\alpha \in \mathcal{F}$, if $h_\alpha(x)$ is the remainder after the univariate division of $f(x)$ by $g(x, \alpha)$, then $h_\alpha(x) \equiv h(x, \alpha)$.
>
> **Proof:** Fix $\alpha \in \mathcal{F}$. Let $f(x) \equiv s(x, y)g(x, y) + h(x, y)$ and $f(x) \equiv s_\alpha(x)g(x, \alpha) + h_\alpha(x)$. We have $\deg_x(h) < \deg_x(g)$ anddeg$(h_\alpha) < \deg(g(x, \alpha)) = \deg_x(g)$. Now, $s(x, \alpha)g(x, \alpha) + h(x, \alpha) \equiv s_\alpha(x)g(x, \alpha) + h_\alpha(x)$, or
>
> $$h(x, \alpha) - h_\alpha(x) \equiv g(x, \alpha)(s_\alpha(x) - s(x, \alpha))$$
>
> If $(s_\alpha(x) - s(x, \alpha))$ is not zero, then the degree of the right hand side is at least $\deg(g(x, \alpha) = \deg_x(g)$ and so must the degree of the left hand side, contradicting what we said before. So, $h(x, \alpha) - h_\alpha(x) \equiv 0$.
> □

Thus, we can represent Q by performing one univariate division for each $\beta = q(\tilde{\beta}) \in L_1$. This algorithm in C++ code is given in Listing 4.

Listing 4: Construction of the bivariate polynomial Q

```
void create_bivariate(vec_GF2EX& bivs, const GF2EX& P,
                      const GF2EX& q, const vec_GF2E& L10_span){
  GF2EX qp;
  GF2E tmp;

  bivs.SetLength(L10_span.length());

  for(long i=0; i<L10_span.length(); i++){ // For each $\tilde{\beta} \in \tilde{L}_1$
    tmp = EvalLinearizedPoly(q, L10_span[i]);
    qp = q - GF2EX(0,tmp);
    bivs[i] =  P % qp;
  }
}
```

Univariate division of two degree d polynomials can be reduced to multiplication of two degree d polynomials using the Sieveking-Kung method (see [vzGG99]); thus, univariate polynomial division can be achieved in $O(d \log d)$ field operations. This is how polynomial division in NTL is implemented. Since we are performing $\sqrt{n}$ divisions of an $n/8$-degree polynomial, we have for this algorithm, $T_f(n) = O(n^{3/2}\log(n))$.

Then, we can rewrite the recurrence for $T(n)$ as:

$$T(n) = \begin{cases} O(n^{3/2}\log(n)) + 2^{\lceil k/2 \rceil} \cdot T(8 \cdot 2^{\lfloor k/2 \rfloor}) + 4 \cdot 2^{\lfloor k/2 \rfloor} \cdot T(2^{\lceil k/2 \rceil}) & \text{if } k > 6 \\ 0 & \text{if } k \leq 6 \end{cases} \tag{1}$$

again with $k = \log(n)$.

Lemma 5: $T(n) = O(n^{3/2}\log n)$.

Proof: We prove by induction that $T(n) \leq c \cdot n^{3/2}(\log n - 6)$ for an appropriate choice of c and for sufficiently large n. For $k > 6$, $8 \cdot 2^{\lfloor k/2 \rfloor} < 2^k$ and hence, we start by assuming that the bound to be proven holds for the recursive calls in (1). For large enough n, there exists a constant d such that:

$$\begin{aligned} T(n) &\leq d \cdot n^{3/2}\log(n) + 2^{\lceil k/2 \rceil} \cdot T(8 \cdot 2^{\lfloor k/2 \rfloor}) + 4 \cdot 2^{\lfloor k/2 \rfloor} \cdot T(2^{\lceil k/2 \rceil}) \\ &\leq d \cdot k \cdot n^{3/2} + c \cdot 2^{\lceil k/2 \rceil} 2^{9/2} 2^{3\lfloor k/2 \rfloor/2}(3 + \lfloor k/2 \rfloor - 6) + 4c \cdot 2^{\lfloor k/2 \rfloor} 2^{3\lceil k/2 \rceil/2}(\lceil k/2 \rceil - 6) \\ &< d \cdot k \cdot n^{3/2} + \frac{c}{2} n^{3/2}\left(\left\lfloor \frac{k}{2} \right\rfloor - 3\right) + \frac{c}{2} n^{3/2}\left(\left\lceil \frac{k}{2} \right\rceil - 6\right) \\ &= n^{3/2}(dk + \frac{c}{2}(k - 9)) \\ &< c \cdot n^{3/2}(k - 6) \end{aligned}$$

The first inequality follows from the defining recurrence relation for $T(n)$ in (1). The second inequality follows from the inductive hypothesis. The third inequality follows from observing that for $k > 22$, $\frac{9}{2} + \frac{1}{2}\left\lfloor \frac{k}{2} \right\rfloor < \frac{k}{2} - 1$ and $2 + \frac{1}{2}\left\lceil \frac{k}{2} \right\rceil < \frac{k}{2} - 1$. The fourth equality is algebra. The fifth inequality follows from having an appropriately large c. As for the base case of the induction, we choose a c so that $c \cdot n^{3/2}(\log n - 6)$ is larger than $T(2^{14})$ and $T(2^{12})$ because these are the values that $T(2^{23})$ depends on. □

So, finding the bivariate polynomial f is the main bottleneck in constructing the PCPP and leads to the rather large running time of the prover in Lemma 5. It remains an open question whether the running time of the prover for this PCPP system can be improved.

2.2.2 Proof Size

As mentioned in the introduction, the size of the PCPP is an important parameter in many applications of the theory. Having a nearly linear proof size has consequences for the construction of locally testable codes, for example. We will show that our PCPPs indeed have this property.

Looking at Listing 3, the proof size[4], $S(n)$, can be recursively characterized as:

$$\begin{aligned} S(n) &= \begin{cases} 2^{\lceil k/2\rceil} \cdot (8 \cdot 2^{\lfloor k/2\rfloor}) + 2^{\lceil k/2\rceil} \cdot T(8 \cdot 2^{\lfloor k/2\rfloor}) + 4 \cdot 2^{\lfloor k/2\rfloor} \cdot T(2^{\lceil k/2\rceil}) & \text{if } k > 6 \\ 0 & \text{if } k \leq 6 \end{cases} \\ &= \begin{cases} 8n + 2^{\lceil k/2\rceil} \cdot T(8 \cdot 2^{\lfloor k/2\rfloor}) + 4 \cdot 2^{\lfloor k/2\rfloor} \cdot T(2^{\lceil k/2\rceil}) & \text{if } k > 6 \\ 0 & \text{if } k \leq 6 \end{cases} \end{aligned} \tag{2}$$

Lemma 6: $S(n) = O(n \log^4 n)$

Proof: We prove by induction that $S(n) \leq c \cdot n \log^4 n$ for an appropriate value of c. We will assume that this bound holds for the recursive calls in (2). Then, we have:

$$\begin{aligned} S(n) &\leq 8n + 2^{\lceil k/2\rceil} \cdot c \cdot 8 \cdot 2^{\lfloor k/2\rfloor} \left(\left\lfloor \frac{k}{2} \right\rfloor + 3 \right)^4 + 4 \cdot 2^{\lfloor k/2\rfloor} \cdot c \cdot 2^{\lceil k/2\rceil} \left\lceil \frac{k}{2} \right\rceil^4 \\ &= 8n + 8cn \left(\left\lfloor \frac{k}{2} \right\rfloor + 3 \right)^4 + 4cn \left\lceil \frac{k}{2} \right\rceil^4 \\ &< 8n + 12cn \left(\left\lfloor \frac{k}{2} \right\rfloor + 3 \right)^4 \\ &< cn \log^4 n \end{aligned}$$

The first inequality is the inductive hypothesis. The second equality is from simplification. The third inequality follows from $\lceil k/2\rceil \leq \lfloor k/2\rfloor + 1$. The fourth inequality holds for large values of k (since $12 < 2^4$). For the base case of the induction, take c to be large enough so that the bound holds for the values of n where the fourth inequality is true. □

Although the proof to the lemma above treats the bounds loosely, the $O(n \log^4 n)$ bound to the solution of the recursion in (2) is pretty tight. In fact, we find from running our program that $S(n) = \frac{1}{4} n \log^4 n$ is a good bound for the proof size.

2.3 The Verifier

The verifier for the Reed-Solomon PCPP uses the bivariate polynomial test analyzed in [PS94] to check that the provided input is indeed close to a Reed-Solomon codeword. All this is done by querying only a constant number of field elements! The test made by the verifier is described in [BSS04] as follows:

Definition 4 ([BSS04], Definition 5) The verifier for proximity to RS($GF(2^\ell)$,L,$d = |L|/8$) receives as input the parameters $GF(2^\ell)$, a basis $(b_1, \ldots, b_k)$ for L and degree parameter $d = |L|/8$. It has oracle access to a purported codeword $p : L \to GF(2^\ell)$ and its purported proof $\pi = \{f, \Pi\}$ and is denoted $V_{RS}^{(p,\pi)}(GF(2^\ell), L, d)$. If $|L| \leq 64$ (in which case $\pi = \emptyset$), the verifier reads p in entirety and accepts iff

[4] We count the number of field elements in the proof. Counting the number of bits leads to another factor of $\log \mathcal{F}$.

$p \in \mathrm{RS}(GF(2^\ell), L, |L|/8)$. Otherwise, it computes $\lfloor k/2 \rfloor$ and performs one of the following two tests with probability half each.

> **Row-Test** Pick random $\tilde{\beta} \in \tilde{L}_1$, set $\beta = q(\tilde{\beta})$, compute basis for $L_{\tilde{\beta}}$ and recursively run $V_{\mathrm{RS}}^{(\hat{f}|_{\beta}^{\leftrightarrow}, \pi_{\beta}^{\leftrightarrow})}(GF(2^\ell), L_{\tilde{\beta}}, |L_{\tilde{\beta}}|/8)$.
>
> **Col-Test** Pick $\alpha \in L_0$ at random, compute basis for L_1 and then recursively run $V_{\mathrm{RS}}^{(\hat{f}|_{\alpha}^{\updownarrow}, \pi_{\alpha}^{\updownarrow})}(GF(2^\ell), L_1, |L_1|/8)$.

In the above definition, $\hat{f}$ is the bivariate function that agrees with the evaluation table of f on S and with $\hat{p}$ on T. Recall from Section 2.2 that $\hat{p}$ is is a partial bivariate function with the partial domain T, defined to be $\hat{p}(x, q(x)) = p(x)$. So, at the top level, when a row or column of $\hat{f}$ is selected, some of its values can be retrieved from querying the bivariate polynomial evaluation table (for f) provided in the PCPP while for others, the input string (the evaluation for p) must be queried. As the verifier gets deeper into the recursion tree, determining where to look in the PCPP for an evaluation of $\hat{f}$ requires looking back at the decision tree of choosing row-tests or column-tests and determining at each level if the needed evaluation of $\hat{f}$ is contained in the bivariate evaluation table at that level. Instead of complicating the implementation of the verifier, it is easier to restructure the proof as an oracle program that automatically determines the correct place to look in itself for an evaluation of $\hat{f}$. Such a program implemented in C++ is shown in Listing 5.

Listing 5: Implementation of a Proof Oracle

```cpp
enum Level {TOP, ROW, COL};

struct verifier_oracle {
  // Evaluation of f on S
  const biv_eval_table* table;

  // Looking at row or column <header> of f
  GF2E header;

  // Pointer to the proof oracle that should be queried for
  // evaluations on T
  const verifier_oracle* parent;

  // If this is the top level, evaluation table of the univariate
  // polynomial p
  const eval_table* orig_poly;

  // The level: top, a row, or a column
  Level lev;

  // The linearized polynomial q
  GF2EX q;

  // Constructor for the top level
  verifier_oracle(const eval_table* orig){
    orig_poly = orig;
    lev = TOP;
    parent = 0;
  }
  // Constructor if this is the row or column projection
  verifier_oracle(const verifier_oracle* par, const biv_eval_table* tab,
                  Level roworcol, GF2E& val, GF2EX& qp){
    parent = par;
    table = tab;
    lev = roworcol;
    header = val;
    q = qp;
  }

  // Recursive query
  GF2E query(const GF2E& ask) const{
    if(lev == TOP)
      return orig_poly->query(ask);

    else if(lev == ROW){
      if(EvalLinearizedPoly(q,ask) != header){
        return table->query(ask,header);
      }
      else{
        return parent->query(ask);
      }
    }
    else{
      if(EvalLinearizedPoly(q,header) != ask){
        return table->query(header,ask);
      }
      else{
        return parent->query(header);
      }
    }
  }
};
```

Using this proof oracle structure, the implementation of the verifier is simple and direct. It is shown below.

Listing 6: Implementation of the PCPP verifier of [BSS04]

```
/** Verify if indeed <proof> is a valid PCPP that shows that <poly>
  * is the evaluation table of a polynomial of degree less than |L|/8.
  */
bool verify_proof(const vec_GF2E& L_bases, const eval_table& poly,
                  const poly_oracle& proof){
  verifier_oracle* root = new verifier_oracle(&poly);
  return verify(L_bases, *root, proof);
}

/** A helper procedure for the above
  */
bool verify(const vec_GF2E& L_bases, const verifier_oracle& oracle,
            const poly_oracle& proof){

```

```
long k = L_bases.length(), index, i;
vec_GF2E L00_bases, L10_bases, L0_bases, L1_bases, Lbeta_bases, L_span;
poly_oracle *rowproof, *colproof;
verifier_oracle* next;
GF2E choice, qchoice;
GF2EX q, poly;
int rand;

// if $k < 7$, simply read in all of the input, interpolate a
// polynomial, and check its degree
if(k < 7){
  get_span(L_span, L_bases);
  eval_table polyvals;

  // maximum of 64 queries here
  for(long i=0; i<L_span.length(); i++){
    polyvals.insert(L_span[i], oracle.query(L_span[i]));
  }

  interpolate_poly(poly, polyvals, L_bases);

  if(deg(poly) < power_long(2,k-3))
    return true;
  else
    return false;
}

else {

  // get the bases for $\tilde{L}_0$, $L_0$, $\tilde{L}_1$ and $L_1$
  get_L00_bases(L00_bases, L_bases);
  get_L0_bases(L0_bases, L_bases);
  get_L10_bases(L10_bases , L_bases);

  LinearizedPoly(q, L00_bases);
  get_L1_bases(L1_bases , L10_bases, q);


  // flip a coin
  if(getRandomBit() == 1){ // check row

    index = 0;
    for(i=0; i<L10_bases.length(); i++){ // choose random element $\tilde{\beta} \in \tilde{L}_1$
      rand = getRandomBit();
      index = index + rand * power_long(2,i);
      choice += rand * L10_bases[L10_bases.length()-i-1];
    }
    // $\beta = q(\tilde{\beta})$
    qchoice = EvalLinearizedPoly(q,choice);

    // get $\pi^{\leftrightarrow}_{\beta}$
    rowproof = proof.proof[index];

    next = new verifier_oracle(&oracle, &(proof.eval), ROW, qchoice , q);
```

```
      get_Lbeta_bases(Lbeta_bases, choice, L_bases);
      // recurse
      return verify(Lbeta_bases, *next, *rowproof);
    }

    else { // check column
      index = 0;
      for(i=0; i<L0_bases.length(); i++){ // choose random element $\alpha \in L_0$
        rand = getRandomBit();
        index = index + rand * power_long(2,i);
        choice += rand * L0_bases[L0_bases.length()-i-1];
      }

      // get $\pi_\alpha^\updownarrow$
      colproof = proof.proof[index + power_long(2, L10_bases.length())];

      next = new verifier_oracle(&oracle, &(proof.eval), COL, choice, q);
      //recurse
      return verify(L1_bases, *next, *colproof);
    }
  }
}
```

The query complexity of the verifier is immediate. The verifier queries at most 64 field elements and, hence, at most $64 \log |\mathcal{F}|$ bits. Next, we look at some other complexity parameters associated with the PCPP verifier.

2.3.1 Randomness Complexity

In [BSS05], it is ascertained that the randomness complexity is $r(k) \leq k + c \cdot \log k$ for a constant c. Here, we give a tighter bound for $r(k)$.

First of all, note that the exact number of coins flipped by the verifier depends on its decision tree of choosing between row-tests and column-tests; this is so because $|L_{\tilde{\beta}}|$ and $|L_1|$ are different for all $\tilde{\beta}$. We want to determine the maximum number of coins that can be be flipped by the verifier, i.e. an upper bound on $r(k)$. Thus, looking at the definition of the verifier, we can write:

$$r(k) \leq \begin{cases} 1 + \max\left(\left\lceil \frac{k}{2} \right\rceil + r\left(\left\lfloor \frac{k}{2} \right\rfloor + 3\right), 2 + \left\lfloor \frac{k}{2} \right\rfloor + r\left(\left\lceil \frac{k}{2} \right\rceil\right)\right) & \text{if } k > 6 \\ 0 & \text{if } k \leq 6 \end{cases}$$

Lemma 7: $r(k) \leq k + 4\lfloor \log(k-6) \rfloor - 1$

Proof: Can be verified immediately through a straightforward induction.

The randomness complexity also allows us a way to bound the proof size, because $S(n) \leq 2^{r(n)} q(n)$ where $S(n)$ is the proof size and $q(n)$ is the query complexity. So, once again, $S(n) = O(n \log^4 n)$.

2.3.2 Running Time of the Verifier

Let $t_V(k)$ denote the running time for the verifier. Then, we have that:

Lemma 8: $t_V(k) = O(k^3)$.

Proof: From inspecting the algorithm given in Proposition 8 of [BSS04], $q(x)$, the linearized polynomial, has k terms and can be computed in time $O(k^3)$. It can be evaluated in time $O(k^2)$. Thus computing the basis for L_1 takes time $O(k^3)$ and similarly for computing the basis for $L_{\tilde{\beta}}$ given $\tilde{\beta}$. Therefore, we can write the following recursion:

$$\begin{aligned} t_V(k) &= O(k^3) + \max(t_V(\lfloor k/2 \rfloor + 3), t_V(\lceil k/2 \rceil)) \\ &= O(k^3) + t_V(\lfloor k/2 \rfloor + 3) \end{aligned}$$

since t_V is monotonically increasing. A simple induction shows that $t_V(k) = O(k^3)$.

3 Conclusion

Our tight bounds on the complexity parameters related to Reed-Solomon PCPPs show that it is indeed feasible in practice to create PCPPs as a semantic analog to error-correcting codes. The question of improving the time performance of the prover remains open.

4 Acknowledgements

I am greatly thankful to Madhu Sudan for introducing me to the theory of probabilistically checkable proofs and for discussing the subject of this paper with me. I have learnt a lot from talking to him and from listening to his understanding of complexity theory.

References

[BSS04] Eli Ben-Sasson and Madhu Sudan. Simple PCPs with Poly-log rate and Query Complexity. Unpublished manuscript, 2004.

[BSS05] Eli Ben-Sasson and Madhu Sudan. Simple PCPs with Poly-log rate and Query Complexity. In *Proceedings of the 37th STOC*, 2005.

[PS94] A. Polischuk and D. Spielman. Nearly-linear size holographic proofs. In *Proceedings of the 26th STOC*, pages 194–203, 1994.

[Sho] Victor Shoup. NTL: A library for doing number theory, version 5.4. http://www.shoup.net.

[vzGG99] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge University Press, New York, NY, USA, 1999.